EXHIBIT A

UNITED STATES DISTRICT COURT

NORTHERN DISTRICT OF CALIFORNIA

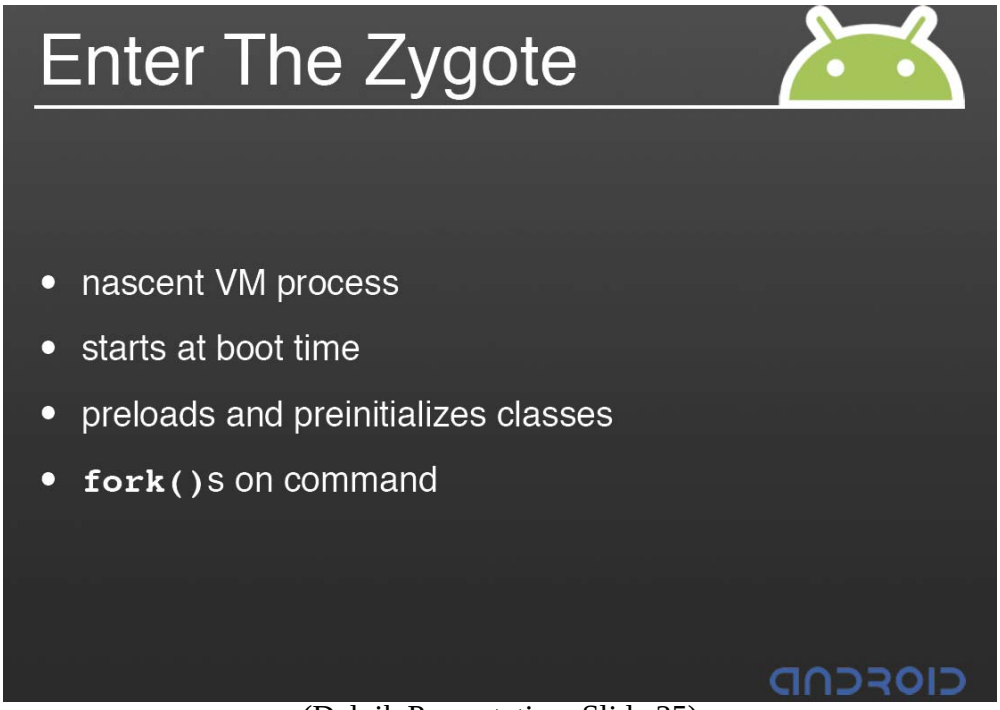
SAN FRANCISCO DIVISION

ORACLE AMERICA, INC. Plaintiff, v. GOOGLE INC. Defendant.	Case No. CV 10-03561 WHA

OPENING EXPERT REPORT OF JOHN C. MITCHELL REGARDING PATENT INFRINGEMENT

SUBMITTED ON BEHALF OF PLAINTIFF ORACLE AMERICA, INC.

REDACTED PUBLIC VERSION



(Dalvik Presentation, Slide 25)

595. And in corresponding Dalvik Video at 13:48:

> "What we do with the zygote, as its name implies,…when it gets a command to start up a new application, it does a normal Unix fork and then that child process becomes that target application. And the result of that is this."

596. The "to clone" limitation of the runtime environment is further described below regarding the copy-on-write process cloning mechanism.

597. The **last element of claim 1** recites a "copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space of the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process." The *zygote* process provides the copy-on-write process cloning mechanism.

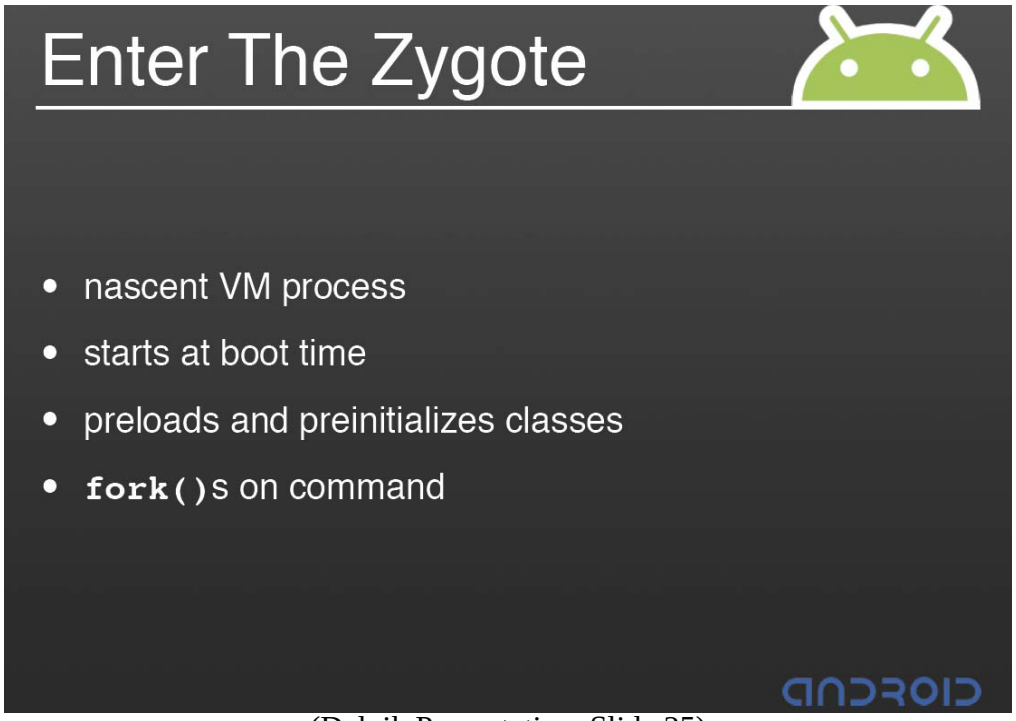
598. Google presentations provide evidence of infringement of this element. The copy-on-write process cloning mechanism is described in the Android Presentation at slide 82.

(Android Presentation, Slide 82)

599. And in corresponding Android Video at 44:30:

> "The init process starts up a really neat process called zygote….It uses copy-on-write to maximize re-use and minimize footprint so that data structures are shared and it won't do a full copy unless some of those data structures are to be modified."

600. The copy-on-write process cloning mechanism is further described in the Dalvik presentation at slides 25-26.

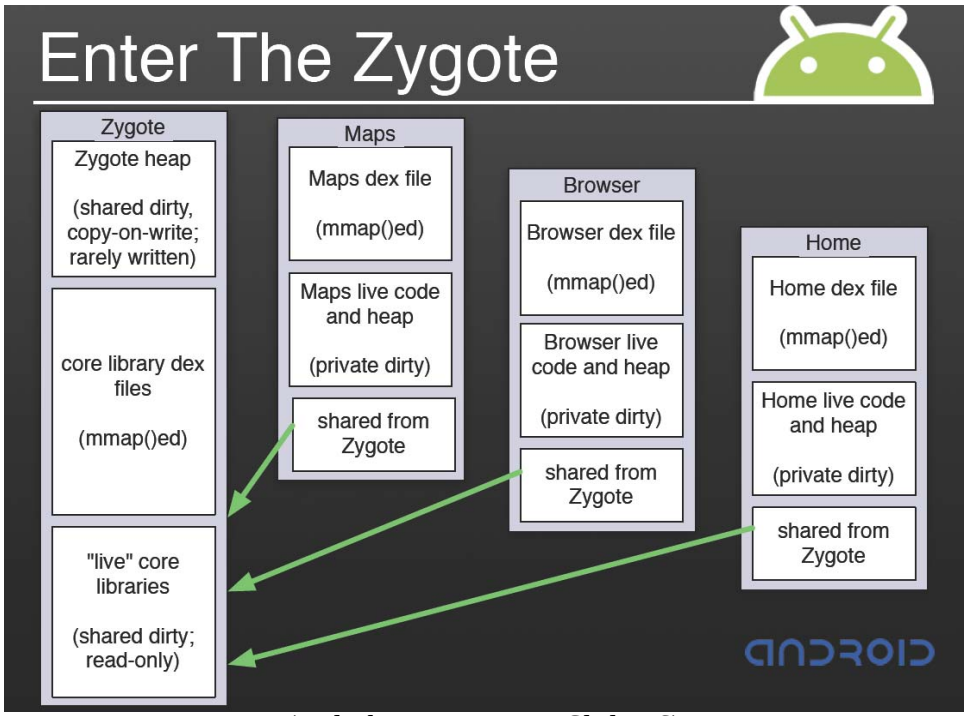


(Dalvik Presentation, Slide 25)

601. And in corresponding Dalvik Video at 13:48:

> "What we do with the zygote, as its name implies,…when it gets a command to start up a new application, it does a normal Unix fork and then that child process becomes that target application. And the result of that is this."

602. The copy-on-write semantics of the Unix fork are described below.



(Dalvik Presentation, Slide 26)

603. And in corresponding Dalvik Video at 14:40:

> "So the zygote, again, has made, has made this heap of objects, it's made this live dex structure and then each application that then starts up, instead of having its own memory for those things, it just shares it with the zygote and also with any other app that's also on the system."

604. The copy-on-write process cloning mechanism is also described as follows at http://developer.android.com/guide/basics/what-is-android.html.

> "Android Runtime
>
> …The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.
>
> Linux Kernel
>
> Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack."

(*See also* 5/4/2011 McFadden Dep. 102:12-103:1, 126:25-128:7; GOOGLE-04-00039541 at 541; GOOGLE-04-00061079 at 079.)

605. The copy-on-write in the Linux fork executed by Android is described as follows in Robert Lowe, *Linux Kernel Process Management* (April 15, 2005), a sample chapter is *available at* http://www.informit.com/articles/article.aspx?p=370047&seqNum=2&rll=1.

"Copy-on-Write
…In Linux, fork() is implemented through the use of copy-on-write pages. Copy-on-write (or COW) is a technique to delay or altogether prevent copying of the data. Rather than duplicate the process address space, the parent and the child can share a single copy. The data, however, is marked in such a way that if it is written to, a duplicate is made and each process receives a unique copy."

606. The Linux fork executed by Android provides the "copy-on-write process cloning mechanism" in its *fork()* system call. Linux provides additional "process cloning mechanisms" in its *vfork()* and *clone()* system calls. These Android Linux process cloning mechanisms are described as follows at http://book.opensourceproject.org.cn/kernel/kernelpri/opensource/0131181637/ch03lev1sec3.html#ch03fig09.

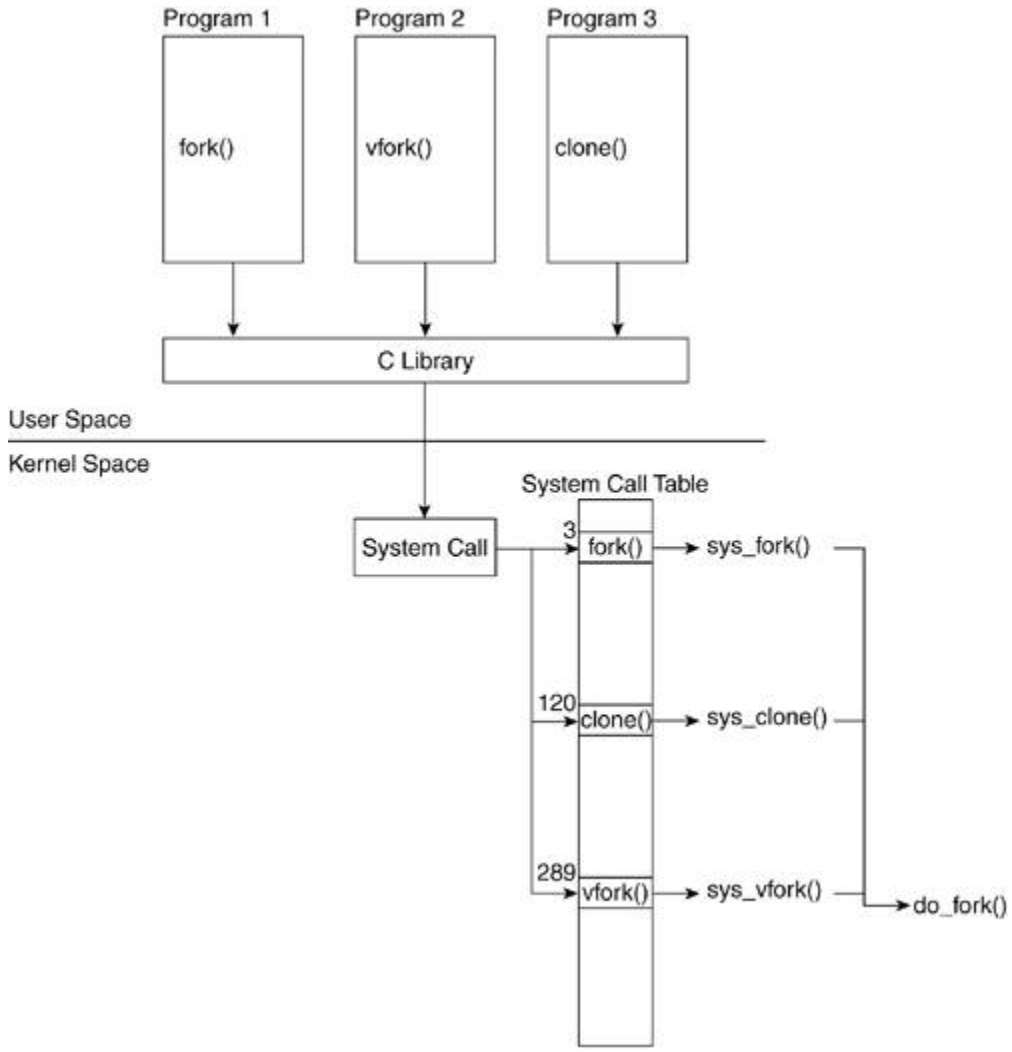
3.3. Process Creation: fork(), vfork(), and clone() System Calls

After the sample code is compiled into a file (in our case, an ELF executable[2]), we call it from the command line. Look at what happens when we press the Return key. We already mentioned that any given process is created by another process. The operating system provides the functionality to do this by means of the `fork()`, `vfork()`, and `clone()` system calls.

[2] ELF executable is an executable format that Linux supports. Chapter 9 discusses the ELF executable format.

The C library provides three functions that issue these three system calls. The prototypes of these functions are declared in `<unistd.h>`. Figure 3.9 shows how a process that calls `fork()` executes the system call `sys_fork()`. This figure describes how kernel code performs the actual process creation. In a similar manner, `vfork()` calls `sys_fork()`, and `clone()` calls `sys_clone()`.

Figure 3.9. Process Creation System Calls



All three of these system calls eventually call `do_fork()`, which is a kernel function that performs the bulk of the actions related to process creation. You might wonder why three different functions are available to create a process. Each function slightly differs in how it creates a process, and there are specific reasons why one would be chosen over the other.

When we press Return at the `shell` prompt, the shell creates the new process that executes our program by means of a call to `fork()`. In fact, if we type the command `ls` at the shell and press Return, the pseudocode of the shell at that moment looks something like this:

```
if( (pid = fork()) == 0 )
   execve("foo");
else
   waitpid(pid);
```

We can now look at the functions and trace them down to the system call. Although our program calls `fork()`, it could just as easily have called `vfork()` or `clone()`, which is why we introduced all three functions in this section. The first function we look at is `fork()`. We delve through the

calls `fork()`, `sys_fork()`, and `do_fork()`. We follow that with `vfork()` and finally look at `clone()` and trace them down to the `do_fork()` call.

3.3.1. fork() Function

The `fork()` function returns twice: once in the parent and once in the child process. If it returns in the child process, `fork()` returns 0. If it returns in the parent, `fork()` returns the child's PID. When the `fork()` function is called, the function places the necessary information in the appropriate registers, including the index into the system call table where the pointer to the system call resides. The processor we are running on determines the registers into which this information is placed.

At this point, if you want to continue the sequential ordering of events, look at the "Interrupts" section in this chapter to see how `sys_fork()` is called. However, it is not necessary to understand how a new process gets created.

Let's now look at the `sys_fork()` function. This function does little else than call the `do_fork()` function. Notice that the `sys_fork()` function is architecture dependent because it accesses function parameters passed in through the system registers.

```
-----------------------------------------------------------------------
arch/i386/kernel/process.c
asmlinkage int sys_fork(struct pt_regs regs)
{
   return do_fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL);
}
-----------------------------------------------------------------------

-----------------------------------------------------------------------
arch/ppc/kernel/process.c
int sys_fork(int p1, int p2, int p3, int p4, int p5, int p6,
             struct pt_regs *regs)
{
        CHECK_FULL_REGS(regs);
        return do_fork(SIGCHLD, regs->gpr[1], regs, 0, NULL, NULL);
}
-----------------------------------------------------------------------
```

The two architectures take in different parameters to the system call. The structure `pt_regs` holds information such as the stack pointer. The fact that `gpr[1]` holds the stack pointer in PPC, whereas `%esp`[3] holds the stack pointer in x86, is known by convention.

[3] Recall that in code produced in "AT&T" format, registers are prefixed with a %.

3.3.2. vfork() Function

The `vfork()` function is similar to the `fork()` function with the exception that the parent process is blocked until the child calls `exit()` or `exec()`.

`sys_vfork()`

```
arch/i386/kernel/process.c
asmlinkage int sys_vfork(struct pt_regs regs)
{
   return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.ep, &regs, 0, NULL,
NULL);
}
-----------------------------------------------------------------------
arch/ppc/kernel/process.c
int sys_vfork(int p1, int p2, int p3, int p4, int p5, int p6,
              struct pt_regs *regs)
{
        CHECK_FULL_REGS(regs);
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs->gpr[1],
                       regs, 0, NULL, NULL);
}
-----------------------------------------------------------------------
```

The only difference between the calls to `sys_fork()` in `sys_vfork()` and `sys_fork()` are the flags that `do_fork()` is passed. The presence of these flags are used later to determine if the added behavior just described (of blocking the parent) will be executed.

3.3.3. clone() Function

The `clone()` library function, unlike `fork()` and `vfork()`, takes in a pointer to a function along with its argument. The child process created by `do_fork()`calls this function as soon as it gets created.

```
[View full width]
-----------------------------------------------------------------------
sys_clone()
arch/i386/kernel/process.c
asmlinkage int sys_clone(struct pt_regs regs)
{
 unsigned long clone_flags;
        unsigned long newsp;
        int __user *parent_tidptr, *child_tidptr;

        clone_flags = regs.ebx;
        newsp = regs.ecx;
        parent_tidptr = (int __user *)regs.edx;
        child_tidptr = (int __user *)regs.edi;
        if (!newsp)
                newsp = regs.esp;
        return do_fork(clone_flags & ~CLONE_IDLETASK, newsp, &regs, 0,
parent_tidptr,
 child_tidptr);
}
-----------------------------------------------------------------------

-----------------------------------------------------------------------
arch/ppc/kernel/process.c
int sys_clone(unsigned long clone_flags, unsigned long usp,
              int __user *parent_tidp, void __user *child_thread\
```

```
ptr,
              int __user *child_tidp, int p6,
              struct pt_regs *regs)
{
        CHECK_FULL_REGS(regs);
        if (usp == 0)
                usp = regs->gpr[1];     /* stack pointer for chi\
ld */
        return do_fork(clone_flags & ~CLONE_IDLETASK, usp, regs,\
 0,
                        parent_tidp, child_tidp);
}
```

As Table 3.4 shows, the only difference between `fork()`, `vfork()`, and `clone()` is which flags are set in the subsequent calls to `do_fork()`.

Table 3.4. Flags Passed to do_fork by fork(), vfork(), and clone()

	fork()	vfork()	clone()
SIGCHLD	X	X	
CLONE_VFORK		X	
CLONE_VM		X	

Finally, we get to `do_fork()`, which performs the real process creation. Recall that up to this point, we only have the parent executing the call to `fork()`, which then enables the system call `sys_fork()`; we still do not have a new process. Our program `foo` still exists as an executable file on disk. It is not running or in memory.

3.3.4. do_fork() Function

We follow the kernel side execution of `do_fork()` line by line as we describe the details behind the creation of a new process.

[View full width]

```
kernel/fork.c
1167   long do_fork(unsigned long clone_flags,
1168       unsigned long stack_start,
1169       struct pt_regs *regs,
1170       unsigned long stack_size,
1171       int __user *parent_tidptr,
1172       int __user *child_tidptr)
1173   {
1174     struct task_struct *p;
```

```
1175     int trace = 0;
1176     long pid;
1177
1178     if (unlikely(current->ptrace)) {
1179        trace = fork_traceflag (clone_flags);
1180        if (trace)
1181          clone_flags |= CLONE_PTRACE;
1182     }
1183
1184   p = copy_process(clone_flags, stack_start, regs, stack_size,
parent_tidptr,
 child_tidptr);
-----------------------------------------------------------------------
```

Lines 1178-1183

The code begins by verifying if the parent wants the new process `ptraced`. ptracing references are prevalent within functions dealing with processes. This book explains only the `ptrace` references at a high level. To determine whether a child can be traced, `fork_traceflag()` must verify the value of `clone_flags`. If `CLONE_VFORK` is set in `clone_flags`, if `SIGCHLD` is not to be caught by the parent, or if the current process also has `PT_TRACE_FORK` set, the child is traced, unless the `CLONE_UNTRACED` or `CLONE_IDLETASK` flags have also been set.

Line 1184

This line is where a new process is created and where the values in the registers are copied out. The `copy_process()` function performs the bulk of the new process space creation and descriptor field definition. However, the start of the new process does not take place until later. The details of `copy_process()` make more sense when the explanation is scheduler-centric. See the "Keeping Track of Processes: Basic Scheduler Construction" section in this chapter for more detail on what happens here.

607. Example source code files that provide the copy-on-write process cloning mechanism include the following.

libcore\dalvik\src\main\java\dalvik\system\Zygote.java,
dalvik\vm\native\dalvik_system_Zygote.c,
linux-2.6\arch\arv32\kernel\process.c,
linux-2.6\kernel\fork.c.

608. Note that the above process.c file is for particular system architecture and is only an example. Other architectures have similar process.c files.

609. An example call chain to various code modules that provide the copy-on-write process cloning mechanism include the following.

forkAndSpecialize calls forkAndSpecializeCommon,
forkAndSpecializeCommon calls fork,
fork call sys_fork,
sys_fork calls do_fork,

do_fork calls copy_process.

610. The file libcore\dalvik\src\main\java\dalvik\system\Zygote.java provides the copy-on-write process cloning mechanism code that declares the module forkAndSpecialize() to link to the native method that performs the cloning process, i.e., the forking, that is "to instantiate the child runtime system process."

```
    /**
     * Forks a new Zygote instance, but does not leave the zygote mode.
     * The current VM must have been started with the -Xzygote flag. The
     * new child is expected to eventually call forkAndSpecialize()
     *
     * @return 0 if this is the child, pid of the child
     * if this is the parent, or -1 on error
     */
    native public static int fork();

    /**
     * Forks a new VM instance. The current VM must have been started
     * with the -Xzygote flag. <b>NOTE: new instance keeps all
     * root capabilities. The new process is expected to call capset()</b>.
     *
     * @param uid the UNIX uid that the new process should setuid() to after
     * fork()ing and and before spawning any threads.
     * @param gid the UNIX gid that the new process should setgid() to after
     * fork()ing and and before spawning any threads.
     * @param gids null-ok; a list of UNIX gids that the new process should
     * setgroups() to after fork and before spawning any threads.
     * @param debugFlags bit flags that enable debugging features.
     * @param rlimits null-ok an array of rlimit tuples, with the second
     * dimension having a length of 3 and representing
     * (resource, rlim_cur, rlim_max). These are set via the posix
     * setrlimit(2) call.
     *
     * @return 0 if this is the child, pid of the child
     * if this is the parent, or -1 on error.
     */
    native public static int forkAndSpecialize(int uid, int gid, int[] gids,
            int debugFlags, int[][] rlimits);
```

611. The file dalvik\vm\native\dalvik_system_Zygote.c provides the copy-on-write process cloning mechanism native code that calls the *fork()* module to perform the copy-on-write cloning that is to "to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process."

612. In the code, the call to *fork()* in the statement "pid = fork()" near the bottom of the code excerpt invokes the Linux fork system call documented above and additionally explained below.

```
/* native public static int forkAndSpecialize(int uid, int gid,
*     int[] gids, int debugFlags);
*/
 static void Dalvik_dalvik_system_Zygote_forkAndSpecialize(const u4* args,
JValue* pResult)
{
     pid_t pid;
    pid = forkAndSpecializeCommon(args);
    RETURN_INT(pid);
}
…
/*
 * Utility routine to fork zygote and specialize the child process.
 */
static pid_t forkAndSpecializeCommon(const u4* args, bool isSystemServer)
{
    pid_t pid;

    uid_t uid = (uid_t) args[0];
    gid_t gid = (gid_t) args[1];
    ArrayObject* gids = (ArrayObject *)args[2];
    u4 debugFlags = args[3];
    ArrayObject *rlimits = (ArrayObject *)args[4];
    int64_t permittedCapabilities, effectiveCapabilities;

    if (isSystemServer) {
        /*
         * Don't use GET_ARG_LONG here for now. gcc is generating code
         * that uses register d8 as a temporary, and that's coming out
         * scrambled in the child process. b/3138621
         */
        //permittedCapabilities = GET_ARG_LONG(args, 5);
        //effectiveCapabilities = GET_ARG_LONG(args, 7);
        permittedCapabilities = args[5] | (int64_t) args[6] << 32;
        effectiveCapabilities = args[7] | (int64_t) args[8] << 32;
    } else {
        permittedCapabilities = effectiveCapabilities = 0;
    }

    if (!gDvm.zygote) {
        dvmThrowException("Ljava/lang/IllegalStateException;",
            "VM instance not started with -Xzygote");

        return -1;
    }

    if (!dvmGcPreZygoteFork()) {
        LOGE("pre-fork heap failed\n");
        dvmAbort();
    }

    setSignalHandler();

    dvmDumpLoaderStats("zygote");
    pid = fork();

    if (pid == 0) {
        int err;
        /* The child process */
….
     } else if (pid > 0) {
   /* the parent process */
```

```
        }
    return pid;
    }
```

613. The file linux-2.6\arch\arv32\kernel\process.c provides a *sys_fork()* code, invoked by the *fork()* call, that calls the *do_fork()* module that is to perform the copy-on-write cloning "to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process."

```
asmlinkage int sys_fork(struct pt_regs *regs)
{
   return do_fork(SIGCHLD, regs->sp, regs, 0, NULL, NULL);
}
```

614. The file linux-2.6\kernel\fork.c provides the *do_fork()* code, that calls the *copy_process()* module, to perform the copy-on-write process cloning "to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process."

```
/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 */
long do_fork(unsigned long clone_flags,
              unsigned long stack_start,
              struct pt_regs *regs,
              unsigned long stack_size,
              int __user *parent_tidptr,
              int __user *child_tidptr)
{
        struct task_struct *p;
        int trace = 0;
        long nr;
…
        p = copy_process(clone_flags, stack_start, regs, stack_size,
                         wake_up_new_task(p, clone_flags);
…
        tracehook_report_clone_complete(trace, regs,
                                        clone_flags, nr, p);
…
        return nr;
}
```